An Integrated Visualization Approach Combining Dynamic Data-Flow Analysis with Symbolic Execution

Laura Troost, Hendrik Winkelmann[a] and Herbert Kuchen[b]

Department of Information Systems, University of Münster, Münster, Germany

Keywords: Data-Flow Analysis, Symbolic Execution, Test Case Generation, Language Server Protocol.

Abstract: Although studies have emphasized that generating test cases with respect to data-flow coverage is a highly effective approach to ensure software quality, there is still a lack of appropriate tooling. We contribute to this by extending the open source dynamic data-flow analysis and visualization tool Dacite with symbolic execution using the open source tool Mulib. Thereby, given a Java program and JUnit test cases, the covered data flow cannot only be identified but the user is able to receive feedback about the data flow not covered by existing test cases and can automatically generate test cases for those. This is especially suited for unit testing and early integration testing. Furthermore, to enhance the comprehensibility the identified data flow is visualized for the user with an integrated visualization using the Language Server Protocol.

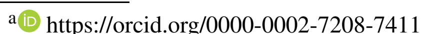
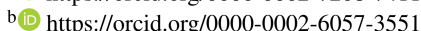
1 INTRODUCTION

Software testing is one of the most widely adopted techniques for assuring high-quality of software systems (Bluemke and Rembiszewski, 2009; Ribeiro et al., 2019; Su et al., 2015). To assess the quality of test cases so that a test suite with higher quality detects more faults, test adequacy criteria are used (Ribeiro et al., 2019). For this, different criteria exist such as control-flow coverage and data-flow coverage (Frankl and Weiss, 1993). While control-flow coverage criteria examine the execution flow of test cases, e.g., measuring which lines were covered, data-flow coverage focuses on the flow of value definitions and usages through the program (Allen and Cocke, 1976; Ribeiro et al., 2019). Even though it has been shown that data-flow-based criteria are more effective than control-flow-based criteria in exposing errors (Frankl and Weiss, 1993; Hemmati, 2015; Ribeiro et al., 2019), control-flow coverage is more commonly used in practice.

One reason for this is the lack of appropriate data-flow coverage tools. Hence, a prototype called Dacite (DAta-flow Coverage for Imperative TEsting) (Troost and Kuchen, 2022) was developed. In contrast to most other tools for data-flow analysis that employ a static analysis, Dacite **dynamically** identifies the reachable data flows of a given Java program and its JUnit test cases. During this execution, it considers challenges such as aliasing and inter-procedural data flows that are hard to solve statically (Troost and Kuchen, 2022). Moreover, it provides an integrated visualization of the identified data flow for common Integrated Development Environments (IDEs) based on the Language Server Protocol (LSP) (Troost et al., 2023). However, as the data flow is derived during the execution, only the passed data flow is identified with no indication of which data flow was not covered yet which is crucial during the test development.

For this reason, this paper proposes the combination of Dacite with the symbolic execution engine Mulib (Winkelmann and Kuchen, 2022). The idea is that by executing the program symbolically, all paths are systematically traversed, and thus, all reachable data flow is derived. This way, the user can receive feedback about which data flow was not covered yet by the existing test cases regarding the coverage criteria *all-uses* which requires the coverage of all definition and usage combinations (Frankl and Weyuker, 1988). The main contributions are:

1. We integrated the data-flow visualization tool Dacite with the symbolic execution engine Mulib.
2. By tracking the data flow during the symbolic execution and comparing it to the data flow covered by existing test cases, it is derived which data flow is not accounted for yet.


[a] https://orcid.org/0000-0002-7208-7411

[b] https://orcid.org/0000-0002-6057-3551

Troost, L., Winkelmann, H. and Kuchen, H.
An Integrated Visualization Approach Combining Dynamic Data-Flow Analysis with Symbolic Execution.
DOI: 10.5220/0012586200003687
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 19th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2024)*, pages 39-50
ISBN: 978-989-758-696-5; ISSN: 2184-4895
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

3. To facilitate comprehensibility the unaccounted-for data flow is visualized to the user adhering to the LSP approach so that the existing IDE integrations are extended with minimal effort.
4. By utilizing input-output mappings which are derived from the symbolic execution, test cases covering the unaccounted-for data flow are automatically generated and suggested to the user.
5. To be able to symbolically execute the program, a driver method initializing the symbolic values is necessary. This is automatically generated and suggested to the user to increase the usability.

Our approach is mainly suited for unit testing and early integration testing involving a small number of classes. For system testing, the symbolic execution might take too long.

After explaining the concepts of data-flow analysis and Dacite in Section 2, Section 3 describes symbolic execution using Mulib. We then describe implementation aspects in Section 4. Therefore, first, the workflow between the user and the components is illustrated. Afterwards, the adaptations necessary to Dacite and Mulib for combining the data-flow analysis with symbolic execution are elaborated. Next, the test case generation and extensions to the visualization are described. In Section 5, the results are evaluated and validated given benchmark examples. Section 6 summarizes the related work concerning data-flow analysis in combination with symbolic execution and Section 7 concludes the results of this paper.

2 DYNAMIC DATA-FLOW ANALYSIS WITH DACITE

Before describing the implementation of combining Dacite's data-flow analysis with the symbolic execution of Mulib, the following sections aim to explain the general concepts of data-flow analysis and the Dacite prototype first.

2.1 Concept of Data-Flow Analysis

The concept of data flow considers the flow of definitions and usages of data through the program. In Dacite, these are represented as definition-usage chains (DUCs) (Troost and Kuchen, 2022; Troost et al., 2023). A DUC relates each variable usage to its most recent definition. In this context, a definition (*def*) is made when a variable receives a new value and a usage (*use*) when this value is referred to (de Araujo and Chaim, 2014; Troost et al., 2023).

Consider the data flow of the program in Listing 1. The program calculates the factorial of a given number.

Listing 1: An implementation of the factorial calculation with the data flow for n.

```
1 public int factorial(int n){
2   int result = 1;
3   int i = 1;
4   while(i <= n){
5     result = result * i;
6     i = i + 1;
7   }
8   return result;
9 }
```

There are 11 DUCs in this program as follows, six chains for the variable `i`, four for the variable `result`, and one for the variable `n` in the form of a triple $(variable, definition, usage)$:

$$\begin{array}{lll}
(i, & \text{int } i = 1, & \text{while}(i <= n)) \\
(i, & \text{int } i = 1, & result = result * i) \\
(i, & \text{int } i = 1, & i = i + 1) \\
(i, & i = i + 1, & \text{while}(i <= n)) \\
(i, & i = i + 1, & result = result * i) \\
(i, & i = i + 1, & i = i + 1) \\
(result, & \text{int } result = 1, & result = result * i) \\
(result, & \text{int } result = 1, & \text{return } result) \\
(result, & result = result * i, & result = result * i) \\
(result, & result = result * i, & \text{return } result) \\
(n, & factorial(\text{int } n), & \text{while}(i <= n))
\end{array}$$

Listing 1 illustrates the reachable data flow for variable `n`. Supposing that the function `factorial` was called externally, i.e., from a class that should not be analyzed, the parameter definitions of this method are considered to be variable definitions (Troost et al., 2023). With the definition in line 1, the usage of variable `n` in line 4 can be reached. Analogously, the data flows of `i` and `result` can be illustrated but were omitted for the sake of clarity.

Since DUCs utilize information on the association of definition and usage, they often are found to be more effective in exposing errors when compared to other control-flow metrics e.g. line coverage (Frankl and Weiss, 1993; Ribeiro et al., 2019). It has been shown that it can increase fault detection up to 79% in comparison to common control-flow metrics (Hemmati, 2015). However, on the other hand, due to its increased complexity, it is more expensive to derive the corresponding information which may be one reason why control-flow metrics are more commonly used in practice (Hemmati, 2015; Ribeiro et al., 2019; Su et al., 2015). Another reason is the lack of appropriate tools for tracking the data flow of a program. Hence, a prototype called Dacite (DAta-flow Coverage for Im-

perative TEsting) was developed (Troost and Kuchen, 2022).

2.2 Dacite

Dacite is an open-source tool[1] that dynamically identifies reachable data flows in Java programs and visualizes them within code editors.

Most existing tools derive the data flow by statically analyzing the program. This approach has some severe limitations in distinguishing reachable and non-reachable DUCs or identifying aliases, i.e., two variables pointing to the same object (Pande et al., 1994; Denaro et al., 2014). To overcome these issues, Dacite dynamically analyzes the program by instrumenting Java programs using the open source framework ASM[2] based on Java Virtual Machine (JVM) bytecode (Troost and Kuchen, 2022). With this, Dacite is able to modify the bytecode of analyzed classes before they are loaded into the JVM (Troost et al., 2023). Consequently, whenever a definition or usage occurs within the program, methods are automatically added to the source code to collect and analyze the data-flow information during the execution. These methods are executed along with the program. All definitions and subsequent usages of variables that were passed during the execution and the relevant information are forwarded to an analyzer class. In this analyzer class, this information is combined to derive passed DUCs (Troost and Kuchen, 2022).

By utilizing this dynamic approach, aliases can be directly identified during the execution. Thus, they are taken into account for the association of a usage to its most recent definition. Additionally, in contrast to other tools, Dacite analyzes the data flow in detail, i.e., it considers DUCs over boundaries of methods and classes and the precise treatment of array elements and object fields (Troost and Kuchen, 2022). Due to the identification of DUCs during the execution, only those DUCs that were passed during the execution are identified. This prevents the static problem of identifying DUCs that are not executable or reachable. However, even though the information of passed DUCs is useful when comparing JUnit test cases, users do not yet receive feedback about DUCs that were not covered by tests. This would be especially valuable in the testing context.

To increase the comprehensibility of the identified data-flow information, the data flow is visualized within the editor during the test development. To mitigate the development effort for implementing different IDE integrations, common and IDE-independent functionalities are extracted into a separate component denoted *language server*. This is then reused for different IDEs using the Language Server Protocol (LSP) which provides a standardized communication protocol via JSON (Microsoft Corporation, 2023a). For Dacite, next to the language server containing functionalities such as executing the analysis and deriving the source code positions of DUCs, there exist two IDE integrations at this point, IntelliJ and Visual Studio Code (VS Code) utilizing this approach (Troost et al., 2023).

Given the factorial example in Listing 1 and the JUnit test case calculating the factorial of 5 (see Figure 1), Figure 2 demonstrates the visualization of Dacite's data-flow analysis in IntelliJ based on this example. The Dacite visualization includes an expandable tree-like list of all identified DUCs covered by the given test sorted by their class, method, and variable. This is shown on the right-hand side of Figure 2 in form of the 10 DUCs covered by the test case. Notice that the DUC `(result, int result = 1, return result)` is not shown, since it is not covered by this test case. To increase the comprehensibility, the identified DUCs can be highlighted within the code editor using the checkboxes. Distinctive colors are utilized to enable the differentiation of variables from each other (see Figure 2 on the left).

3 SYMBOLIC EXECUTION WITH MULIB

Symbolic execution is a well-known technique for automated software verification and test case generation (Cadar and Sen, 2013). Mulib is an open-source symbolic execution engine for programs compiled to Java bytecode (Winkelmann and Kuchen, 2022).[3] It is capable of handling symbolic primitive values and symbolic arrays (with primitive or reference-typed elements) (Winkelmann and Kuchen, 2022; Winkelmann et al., 2021; Winkelmann and Kuchen, 2023). In the following, first, it is described how symbolic execution works and how it can be used to generate test cases. Thereafter, the concept of driver methods is explained in the context of Mulib.

3.1 Symbolic Execution for Test Case Generation

In symbolic execution (Cadar and Sen, 2013), variables and fields are not limited to holding concrete values, such as, e.g. the value `int i = 42;`. Instead,

[1] https://github.com/dacite-defuse/DynamicDefUse

[2] https://asm.ow2.io

[3] https://github.com/NoItAll/mulib

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

```
1  package tryme;
2
3  import org.junit.Test;
4  import static org.junit.Assert.assertEquals;
5
6  public class FactorialTest {
7      @Test
8      public void testSort() {
9          Factorial f = new Factorial();
10         assertEquals( expected: 120,f.factorial( n: 5));
11
12     }
13 }
```

Figure 1: Screenshot of the JUnit test case used to execute the Dacite Analysis.

```
1  package tryme;
2
3  public class Factorial {
       4 usages
4      public int factorial(int Defn) {
5          int Defresult = 1;
6          int Defi = 1;
7          while(Usei <= Usen){
8              Defresult = Useresult * Usei;
9              Defi = Usei +1;
10         }
11         return Useresult;
12     }
13 }
```


Figure 2: Exemplary screenshot of the Dacite visualization in IntelliJ based on given test.

symbolic values are allowed. Symbolic values are not limited to a concrete value and rather describe a whole domain of values. During symbolic execution these domains are restricted via constraints. Constraints, on the other hand, are generated, e.g., whenever a symbolic value is part of the condition of a conditional jump. Again, consider the iterative formulation of computing a factorial known from Listing 1. A potential symbolic value might be the parameter `n`. For each evaluation of the while-condition `i <= n`, (line 4), a *choice point* with two *choice options* is created. Here, the constraints would be `1 <= n` and `1 > n`, where `n` holds the symbolic value, since it is the first iteration of `factorial`. Such a choice option is a representation of *nondeterminism*: The first choice option considers the case where `1 <= n` is true. If this choice option is evaluated, first, the new constraint is pushed onto a constraint stack, and the constraint stack is evaluated in terms of its satisfiability. If the constraint system is found to be satisfiable, symbolic execution enters the body of the while-loop (lines 5 and 6) assuming in the following that the symbolic value of `n` is larger than or equal to `1`. It then would loop back, check the while-condition (line 4) and encounter another choice point with the choice options `2 <= n` and `2 > n`. Consider the case where now the second choice option with the constraint `2 > n` is assumed to be true. In this instance, symbolic execution would not enter the while-loop and instead return the result.

When symbolic execution terminates, it is possible to assign concrete values to the symbolic values by picking a value that satisfies all constraints using the constraint solver. For the exemplified constraint stack $1 \leq n \wedge 2 > n$, the value 1 would be assigned to n, yielding an input-output mapping of $n = 1, return = 1$. Such an input-output mapping can then be transformed into an executable JUnit test case. After completing this instance of execution, symbolic execution checks for other, unevaluated choice options, here, `1 > n` or `2 <= n`. It backtracks and resets the constraint system to the state when encountering the respective constraint and, if the new constraint stack is satisfiable, spawns a new execution instance. This execution would then assume the respective constraint to be true. If the accumulated constraints are not satisfiable, the current execution is discarded and a remaining choice option is tried.

The decision on which choice option to evaluate next is subject to a search algorithm, such as, for example, Depth-First Search (DFS), Breadth-First Search (BFS), or Iterative Deepening Depth-First Search (IDDFS), and potential budgets, such as a time budget, a maximal number of choice points to consider during symbolic execution, etc.

In conclusion, symbolic execution can be used to systematically traverse the execution paths of a Method Under Test (MUT). The initial downside of Dacite, mentioned in Subsection 2.2, can be rectified by a symbolic execution that finds the reachable DUCs. In theory, by systematically traversing all executable paths, Mulib is able to find all reachable DUCs. However, in case that Mulib is stopped due to, e.g., an exhausted (time) budget, it may happen that not all traversable paths are traversed and some DUCs are not found. Nevertheless, in practice, Mulib offers developers important insights into uncovered behavior (see Section 5).

3.2 Driver Methods for Symbolic Execution

To generate test cases for an MUT using symbolic execution, Mulib encloses it in a *search region* via a so-called *driver method* (Winkelmann et al., 2022). A search region comprises all code that shall be symbolically executed. The driver method sets up the, potentially symbolic, inputs, executes the MUT using these inputs, and records the output of the MUT. Consider Listing 2 for an exemplary driver method.

Listing 2: A driver method for initializing the context in which an MUT test is called.

```
1 static int driverFactorial() {
2   Factorial fac = new Factorial();
3   Mulib.remember(fac, "arg0");
4   int n =
5     Mulib.rememberedFreeInt("arg1");
6   int result = fac.factorial(n);
7   return result;
8 }
```

In the driver, the inputs with which the MUT is called are constructed. Since `factorial` is not static, an object of the class defining the method, here `Factorial` is generated (line 2). This object is *remembered* in its current state (line 3). If `factorial` had a state that would be mutated by the MUT, the state before executing `fac.factorial(n)` could be retrieved later on, using the name it was remembered by. This functionality is needed in order to keep track of previous values in case of *destructive updates* (Visser et al., 2004). Thereafter, a symbolic value of type `int` is generated (lines 4 and 5) and used to call the MUT (line 6). Afterwards, the result is returned from the driver method. More details on the inner workings of Mulib are given in Subsection 4.2.

Such driver methods can be custom-tailored to the search region at hand. For instance, symbolic values can be restricted to reduce the run time of the symbolic execution. Consider, for instance, Listing 1: If `n` is not restricted, the loop in lines 4–7 potentially is executed indefinitely. By restricting `n` to, e.g., to all values $n < 5$, the search space is pruned. Furthermore, complex input objects can be constructed while accounting for their invariants and potentially restricting only parts of the inputs to be symbolic. In consequence, if driver methods are being restricted, this must be done carefully to still allow for discovering all DUCs.

For the given example, a call to `Mulib.getPathSolutions(Factorial.class, "driverFactorial", mb)` executes this driver method until either a budget has been reached or all possible choice options have been evaluated. `mb` is a configuration builder in which various budgets can be set. In Mulib, the remembered values, as well as the return value are contained in `Solution` objects which are part of the output.

4 IMPLEMENTATION

To be able to derive which DUCs are not covered yet by given JUnit test cases regarding the coverage criteria *all-uses* which requires the coverage of every existing DUC (Frankl and Weyuker, 1988), the dynamic data-flow analysis of Dacite needs to be combined with the symbolic execution of Mulib. Moreover, based on the symbolic execution, input and output values can be determined for an MUT which can be used for the test case generation. Consequently, for each identified unaccounted-for DUC, a test case can be generated covering this chain. This test case then is suggested to the user.

To facilitate these functionalities, both tools need to be adapted to be able to interact with one another. Subsection 4.1 first presents the interaction workflow between the user, Dacite, and Mulib. Afterwards, given the concepts of Mulib in Section 3, Subsection 4.2 outlines the necessary adaptations and extensions of Mulib in more detail. Then, the adaptations to Dacite and the visualization for the derived DUCs are elaborated in Subsection 4.3 while Subsection 4.4 provides more details on the test case generation.

4.1 Workflow

As mentioned before, Dacite and Mulib need to interact in order to derive unaccounted-for DUCs. Moreover, the user interacts with Dacite via its IDE integration e.g. into Intellij IDEA. Figure 3 illustrates this interaction.

After using Dacite to identify all passed DUCs as shown in Figure 2, the user can start the process of deriving the unaccounted-for DUCs by pressing a corresponding button. To be able to execute the program symbolically, a driver method is required to set up the symbolic inputs, execute the MUT, and record the output (see Subsection 3.2). Hence, when the user starts the process, first a driver method is automatically generated by Dacite and shown to the user. This allows the user to make adaptions to the driver. For instance, the domains of values might be restricted, e.g., the length of a symbolic array might be limited.

After the user has finished adapting the driver method, they can press a button to start the symbolic execution. For this, Dacite first instruments the program classes to be able to derive DUCs during the symbolic execution as described in Subsection 2.2 for the dynamic analysis analogously. Afterwards, the Mulib process of symbolic execution is triggered with the given driver method. For this purpose, Mulib first loads the classes with the Dacite instrumentation and transforms them to enable symbolic execution. Instructions from Dacite are not transformed (see Subsection 4.2). Then, the classes adapted by both Dacite and Mulib are executed. During this process due to the Dacite instrumentation, DUC information is forwarded to the analyzer class whenever a usage or definition occurs. This information is utilized to de-

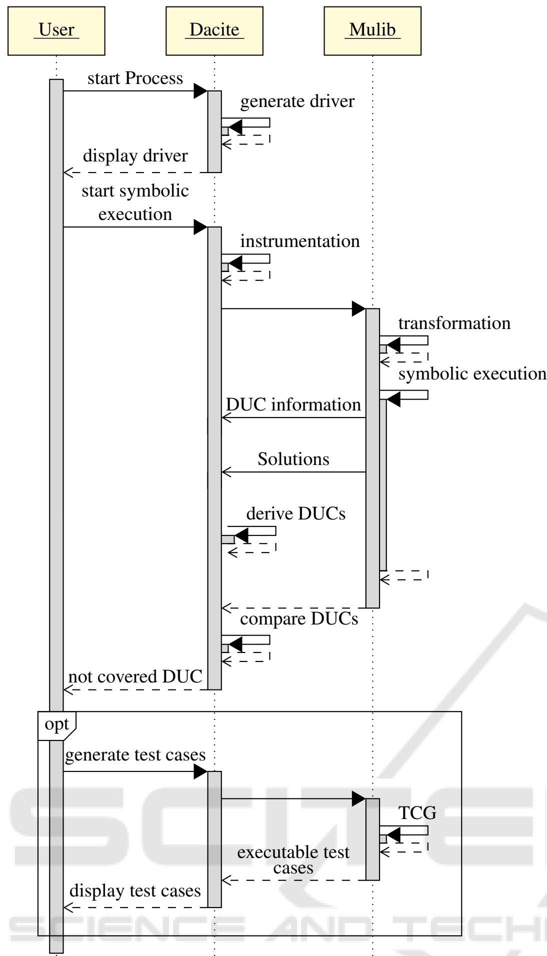


Figure 3: Sequence diagram illustrating the interaction between the user and the components, Dacite and Mulib.

rive the corresponding DUCs. Moreover, `Solution` objects containing concrete input and output values for the current execution path are forwarded to the analyzer class whenever the symbolic execution has reached a leaf node of the search tree. This information is necessary for the test case generation later on and is related to the identified DUCs on this path. This process of the symbolic execution is independent of the previously executed dynamic analysis of Dacite as described in Subsection 2.2 to derive the passed DUCs. However, when the symbolic execution finishes, Dacite compares the symbolically derived DUCs with the passed DUCs from the original analysis to derive differences, i.e., DUCs that were not passed by the given JUnit test cases. These DUCs are then forwarded and displayed to the user.

Lastly, after receiving feedback about the DUCs that are not covered yet by the given JUnit test cases, the user has the option to receive automatically generated test cases for them. Therefore, Dacite displays a button which when pressed by the user invokes the corresponding methods in Mulib to generate test cases for the given `Solution` objects. The executable test cases are returned to Dacite and displayed to the user in a new class. In our running example, a JUnit test case for `factorial(0)` would be generated with expected result `1`. This test case covers the last and currently uncovered DUC $(result$, int $result = 1$, return $result)$.

4.2 Adaptations to Mulib

Since the JVM does not account for symbolic values by itself, Mulib transforms the driver method and all code used within it and generates *partner classes* for the classes used in the search region (Winkelmann and Kuchen, 2022). For this, similar to Dacite, Mulib uses the bytecode library Soot (Vallée-Rai et al., 2010). A partner class represents the original class in the search region and has been adapted so that symbolic execution is accounted for. For instance, calls to *indicator methods*, such as `Mulib.rememberedFreeInt(...)` are replaced with method calls that return a `Sint`-typed value. `Sint` is a library class of Mulib that can represent symbolic expressions. These library types reference the symbolic execution engine of Mulib which is capable of performing search as was exemplified in Subsection 3.1. For instance, instead of comparing two integer numbers, e.g., $i0 < i1$, a method is invoked that checks whether there is a symbolic value involved in the if condition and if there is, which choice point should be evaluated, e.g., `i0.ltChoice(i1, se)`. `se` here is a facade for contacting the symbolic execution engine. For more details, see (Winkelmann and Kuchen, 2022). One particularity of Mulib is that, during this program transformation, special cases can be defined that are exempt from said transformation. Mulib thus is able to ignore method calls added by the instrumentation of Dacite while transforming the remaining search region. While, in the past, approaches to record DUCs in symbolic execution relied on a very tight integration between the symbolic execution engine and the DUC analyzer (Winkelmann et al., 2022), by excluding the functionality of an instrumentation-based stand-alone DUC analyzer from the program transformation in Mulib, minimal adaptations are necessary.

For the purpose of *remembering* objects at a specific state, a new type of constraint, a `PartnerClassObjectRememberConstraint` has been implemented. There are two cases. Either we know the contents of a (implicitly) remembered array or other object, or it is symbolic and not yet lazily initialized. Lazy initialization is a technique for symbol-

ically representing objects (Khurshid et al., 2003): When symbolically initializing an object, the fields of this object are set to symbolic values. In consequence, object-typed fields are set to symbolic object references that, in turn, are initialized symbolically. Since class graphs oftentimes have circular dependencies, generating a symbolic instance of an object with an object-typed field might lead to a an endless recursion. To circumvent the eager initialization of the respective circular object graph, fields are only initialized when a field access occurs. In consequence, if a remembered object either was initialized by executing a constructor, or is symbolic and was already lazily initialized, a deep-copy of it is stored in the aforementioned constraint. If, while executing the MUT, the object is altered, a copy of the state before executing the MUT can still be used for recreating the inputs for a test case. However, if the object is symbolic and was not already lazily initialized, we instead store an identifier for the object and record all subsequent accesses such as loads from an array or retrieving a value from an object's field. Loads of reference-typed values are represented by loading the identifier of the element (see (Winkelmann and Kuchen, 2023) for more information). These accesses, up until a write access, are used to label the object, thus recreating its initial state.

Finally, callback methods were added to Mulib: Every time the driver method is exited, either because a result was found, a budget was exceeded, or we backtrack due to the chosen search strategy, we provide the option to specify a callback. This is required to deal with *potential DUCs*, as is explained in Subsection 4.3.

4.3 Adaptations to Dacite's Analysis

To support the process depicted in Figure 3, two adaptions are necessary for Dacite's analysis, first, the generation of the driver method and second, the handling of the symbolic execution.

Concerning the former, the symbolic execution with Mulib expects a driver method as shown in Listing 2. However, developing such a method requires domain knowledge about the methods of Mulib and its symbolic execution. To facilitate users without this knowledge to derive which DUCs are not covered by the given JUnit test cases, this driver method is automatically generated by Dacite. For this, a static analysis using ASM was carried out based on a specified test case in order to derive the MUT, its input and output values, and their types. Methods called in this test case that stem from classes within defined packages are considered to be MUTs and are extracted. With this information, driver methods are automatically generated containing a default setup initializing the symbolic inputs that can be adapted if needed (see Subsection 3.2).

The second adaptation of Dacite concerns how DUCs are derived during symbolic execution. First, in contrast to the original dynamic analysis the instrumented `.class`-files are saved so that Mulib will load the already instrumented classes instead of the original ones. Afterwards, a call to `Mulib.getPathSolutions(...)` starts the symbolic execution within Mulib. During this, the DUC information is collected within the analyzer class. To derive which definition belongs to a variable usage next to the variable name and its index with which it is stored within the bytecode variable table, the variable value is utilized. The runtime value of a variable is necessary to uniquely map a usage to its definition.

When dealing with symbolic values, assigning a variable usage to its last definition is not always directly possible as it is not clear whether the definition and usage concern the same value and thus, form a DUC. For instance, consider two symbolic variables of the class `Sint`, $n1$ and $n2$. Both have a set of constraints e.g. $n1 < 2$ but no concrete values. One possible scenario is that after resolving the constraints after terminating an execution path, $n1$ and $n2$ have the same value e.g. 1. But it is also possible that they have a different value, e.g. $n1 = 1$ and $n2 = 0$. This poses the challenge that during the symbolic execution, not all variables can be immediately compared to derive the corresponding DUCs. Hence, the DUCs for those variables that cannot be compared can only be derived at a later point when the symbolic values have been labeled to concrete ones. Hence, a different handling has to be implemented for such values denoted here `potential DUCs`. Therefore, instead of directly relating a usage to its definition as for concrete values (for more detail see (Troost and Kuchen, 2022)), symbolic usages and definitions are collected separately. Only when a leaf node of the symbolic execution is reached, the analyzer is called via a callback method of Mulib (see Subsection 4.2) to resolve the symbolic values, map each usage to its last definition, and store the generated DUCs. Furthermore, for all DUCs derived at this execution path, a `Solution` object including the input and output values for this path is added to account for the test case generation later on. So for each DUC, a set of `Solution` objects can be derived as a DUC can be covered by more than one combination of input and output values and its corresponding execution path. Moreover, vice versa, for each `Solution` object it can also be derived which DUCs are covered with this which is necessary for the test case reduction later on (see Subsection 4.4).

The result of the analysis is forwarded as an XML list of derived DUCs and `Solution` objects to the language server (see Subsection 2.2). There, it is compared which DUCs were already identified during the analysis of executing the existing JUnit tests. The difference, i.e., the DUCs which are not covered by the existing tests, are stored and displayed to the user.

4.4 Test Cases Generation

After symbolically executing the program, the user can trigger the test case generation for the DUCs not covered by the given test cases (see Figure 3). This is performed by Mulib's test case generator.

This test case generator transforms input-output-relations into a String representation of an executable JUnit test class calling the MUT with the specified input and comparing the output to the expected one from the relation. Before any String is generated, the number of test cases is reduced. We strive for a lossless reduction with regards to some metric. While Mulib itself is capable of collecting branch coverage as a metric, in the given integration, the metric is the *all-uses* coverage (Frankl and Weyuker, 1988), i.e., all DUCs that have not yet been covered by the pre-existing test set should be covered.

To derive the test cases before the reduction, all distinct `Solution` objects are collected from those *unaccounted-for* DUCs, i.e., DUCs that were not covered using the initial test set. For each Solution object, the set of *unaccounted-for* DUCs which were covered in the path of this solution or test case is derived. Thus, the coverage is represented as a `BitSet` where each set bit represents the unique identifier of one those *unaccounted-for* DUC. Consequently, two test cases covering the same DUCs have the same BitSet and hence, can be reduced to one test case. We search for a minimal subset of test cases so that the number of set bits and thus, the number of covered DUCs, is equal to the number of set bits in the overall set of test cases. Since this is an instance of the NP-hard set coverage problem, typically heuristics are employed (Majchrzak and Kuchen, 2009). We employ a configurable set of heuristics which are described in the following.

1. The first strategy sequentially adds test cases to a result set of the to-be-added test case increases the cardinality of the overall bit vector (Winkelmann et al., 2022).
2. The second strategy starts with a set of test cases where each test case is included. Then, test cases are iteratively removed if the overall cardinality of the remaining test cases did not decrease (Winkelmann et al., 2022).
3. The third strategy starts out with an empty set of test cases. It then greedily adds the test case that, for the current result set of test cases, offers the maximum increase in cardinality when added to the result set.

These strategies offer a low run time and can be combined either sequentially or compete with one another. It is, for instance, possible, to first reduce a test set via the first strategy and then forward the output to the second strategy. It is also possible to specify that multiple (combinations of) strategies should compete with one another, where the smallest test set is chosen.

4.5 Extensions to the Visualization

As mentioned in Subsection 2.2, Dacite's visualization is based on the Language Server Protocol (LSP) to mitigate the implementation effort for developing integrations for different IDEs. In order to extend the visualization to unaccounted-for DUCs and test case generation, it is desirable to adhere to the existing architecture (language server and client) and LSP standard message types (Microsoft Corporation, 2023b).

There are three different user interactions and visualizations necessary as depicted in Figure 3. First, the user needs to start the process of generating a driver method for the symbolic execution. This is displayed by the language client, e.g. IntelliJ, as a button within the tool window next to the list of covered DUCs. This triggers the driver generation for the current test case. After the generation, the driver method within a driver class is created as a new file and is displayed to the user.

Afterwards, the user requires an option for starting the symbolic execution when they are finished adapting or inspecting the driver method. This can be displayed similarly to Dacite's analysis trigger with a LSP request `Code Lens` (Troost et al., 2023). For these requests, the language server determines whether the opened file the user is inspecting adheres to the naming standard as described above and hence, contains one or more driver methods. If that is the case, the corresponding button is displayed to the user by the client. The derived DUCs not covered by the existing tests are visualized analogously to the covered DUCs as depicted in Figure 2. This gives the user a structured collapsible list of the derived unaccounted for DUCs sorted by the class, method, and variable the value definition occurred for. To increase the comprehensibility of this list, the DUCs can be highlighted within the source code as well using the color *red* to distinguish the DUCs covered and not covered by the given tests. Figure 4 illustrates these extensions for the IDE IntelliJ IDEA. Given the facto-

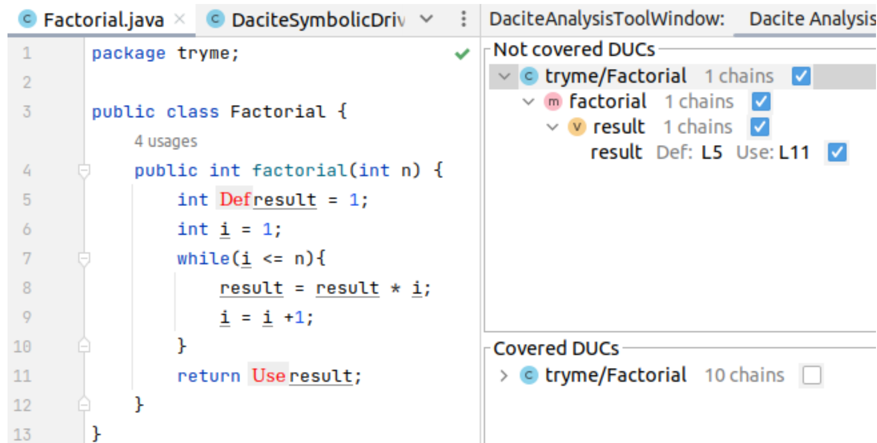


Figure 4: Exemplary screenshot of the Dacite visualization extensions in IntelliJ.

rial example in Section 2 and a JUnit test calculating the factorial for the number five, the symbolic execution was executed with the corresponding driver (see Listing 2). Based on this test, one DUC can be identified which is not covered yet: $(result$, int $result = 1$, return $result)$. This is returned by the symbolic execution and can be seen on the right above the list of covered DUCs in Figure 2. Using the checkbox, the corresponding DUC is highlighted within the editor.

Lastly, the user requires an option to generate test cases for the unaccounted-for DUCs. This is displayed by the language client analogously to the trigger for the driver generation as a button within the tool window which creates a new class containing the generated test cases. By adhering to the existing architecture and LSP requests, both IDE integrations, IntelliJ and VS Code could be extended by these features as shown exemplary in Figure 4 without much further effort.

5 EVALUATION

In order to evaluate and demonstrate the derivation of unaccounted-for DUCs based on the symbolic execution with Dacite and Mulib, this approach was executed on a set of different examples. The majority of examples are retrieved from the SV-COMP set of software verification which is a publicly available benchmark suite for verification and validation software tools released from the annual competition for software verification SV-COMP (Beyer, 2021).

Different types of algorithmic challenging examples were selected, e.g. recursive, sorting, searching, and algorithms exhibiting large sequences of case distinctions, to demonstrate the wide applicability of this approach. Next to the example presented in Section 2 and the algorithms for the Euclidean greatest common divisor, ten examples were retrieved from SV-COMP. Besides different smaller well-known benchmarks such as the recursive Fibonacci algorithm, greatest common divisor, insertion sort, and recursive algorithms checking whether a number is even or odd or whether it is a prime number, we have also tested larger and more complex examples. An implementation of the traveling salesperson problem, a wheel brake system, which determines based on the environment how much brake pressure should be applied, an implementation of a complex alarm system and the implementation of an infusion manager system are utilized. Due to the exponential increase in possible paths during symbolic execution, Glass-box test case generation based on the code is mainly suited for unit testing and for testing a small number of classes in combination (Cadar and Sen, 2013). Hence, we have focused on such scenarios in our experiments.

For each example, a basic JUnit test case was added as a starting point for deriving DUCs that are not accounted for by the given test e.g. considering the factorial example in Listing 1 a test case executing the algorithm for the number five was added (see Figure 1). Based on this, a driver method was automatically generated using the approach explained in Section 4. This can already be utilized to retrieve results from the symbolic execution. However, in order to generate better results, this driver method was adapted. The set of examples and adapted driver methods too are open source[4].

Moreover, in order to avoid an enormous or even infinite number of iterations of the symbolic execution especially for the more complex examples, the execution time was restricted to 10 seconds. Note that these 10s encompass the complete symbolic ex-

[4]https://github.com/dacite-defuse/examples

Table 1: Executed examples with the LOC, the minimal/maximal run time in seconds, the number of covered DUCs, the number of unaccounted-for DUCs by the initial test case, and the number of automatically generated and reduced test cases covering these unaccounted-for DUCs with a symoblic execution limit of 10s.

Example	LOC	Min/Max run time in s	Covered DUCs	Unaccounted-for DUCs	Reduced Test Cases
Factorial	12	1.92/2.07 s	10	1	1
Fibonacci	14	1.93/2.19 s	2	2	1
EuclidianGcd	16	1.93/2.15 s	14	4	3
InsertionSort	17	1.92/2.20 s	36	6	2
RecursiveGcd	22	2.08/2.22 s	2	9	2
EvenOdd	27	2.00/2.16 s	3	3	1
Hanoi	35	2.40/2.77 s	5	3	1
Prime	50	1.96/2.52 s	3	0	0
TspSolver	78	5.20/5.52 s	21	34	2
WBS	241	2.75/2.95 s	41	55	5
Infusion	692	18.99/19.61 s	25	194	34
Alarm	1378	17.68/18.45 s	169	79	10

ecution started by Mulib until the derived DUCs are identified (see the workflow of Mulib in Figure 3). Although the execution time can oftentimes be reduced by providing more complex driver methods, i.e., restricting the input space by employing knowledge on the program, 10 seconds has been found to be both acceptable in practice as well as covering many unaccounted-for DUCs. As a search strategy, we retrieve new choice options in an IDDFS fashion: a choice option with the lowest depth in the search tree is navigated to using symbolic execution. Then, the search region is executed until either a budget is reached or a path solution is found. To assure termination, the exploration of the search tree has been limited to a depth of 64. This limit has been determined via experimentation.

Table 1 presents the results. For each example, the number of executable Lines Of Code (LOC) excluding empty lines and comments was derived This serves as an indicator of the method's complexity. Furthermore, the minimal and maximal overall execution time of symbolic execution out of five different executions in seconds is given for each example. The complete process of generating the driver and executing the symbolic execution until the test case generation is dependent on several user interactions (see Figure 3) which impedes comparable run time measurements. The time displayed in the table was measured from the moment the user starts the symbolic execution (after the driver generation) until the delta of DUCs which were not covered by the initial test case is derived and visualized to the user (the second process in Figure 3). This incorporates the main and most expensive computations (the instrumentation by Dacite, symbolic execution by Mulib, and transformation and visualization of the identified data flow) and thus, presents a sufficiently comparable benchmark. In addition, the number of DUCs covered by the initial JUnit test case, the number of DUCs derived from the symbolic execution that are not accounted for by the initial test case, and the number of JUnit test cases, that were automatically generated and reduced, covering those DUCs is also displayed for each example.

All execution times range in seconds while the execution time of the majority of examples remains under the configured limit of 10s for the symbolic execution. This shows that the symbolic execution for these examples was not aborted in the middle due to this limitation. For the two larger examples (*Infusion* and *Alarm*), in addition to the results in Table 1 the symbolic execution was executed again with a higher limit of 120 s to make sure that no DUCs were missed due to the budget. The example *Infusion* did not identify more DUCs with more time leading to the assumption that all reachable DUCs were already identified. As the Infusion but especially the Alarm example is considerably larger and more complex in terms of conditional statements than typical MUTs subject to unit testing, it is reasonable that it requires more execution time than the other examples. Increasing the symbolic execution limit to 15 min has yielded that the Alarm example has 253 DUCs, thus 84 unaccounted-for DUCs. However, only a small amount of time already resulted in the identification of the majority of unaccounted-for DUCs which would improve the test coverage.

The results demonstrate that our approach is able to identify missing DUCs for each example and is able to generate tests covering these. It can be seen that for the example `Prime` no further DUCs have been identified. As the example given by SV-COMP

contains an endless loop, only the already identified DUCs are reachable. This further demonstrates that the symbolic execution by Mulib is able to derive only those DUCs that can be executed by the program. To validate if all unaccounted-for DUCs were identified, the number of overall DUCs was derived for the smaller examples manually due to the lack of a comparable data-flow analysis tool. For the more complex examples, deriving all possible DUCs per hand is not feasible as these consist of hundreds of chains (see Table 1). However, due to the advanced visualization and source code highlighting, provided by Dacite, the data flow can be comprehended, and missing chains are identified by a generated annotation. The generated test cases were validated by executing them with the data-flow analysis in combination with the initial test cases to demonstrate that all unaccounted-for DUCs are then covered.

With the scope of symbolic execution in mind, we deem the results acceptable and usable in practice, in particular for unit testing and early integration testing. Our approach provides the largest benefit when applied to algorithmically challenging applications, since for such applications it is hard to reach data-flow coverage by manually developed test cases.

6 RELATED WORK

There exist few approaches to aiding the data-flow analysis by combining it with symbolic execution. The majority focus on a static data-flow analysis which implies the limitation that the identified DUCs may not be reachable within the program (see Subsection 2.2). For this, Arzt et al. (Arzt et al., 2015) developed a tool for Java programs denoted TASMAN as a separate post-analysis step based on symbolic execution to overcome this limitation. Via symbolic execution, the tool scans for contradictions on a data-flow path and thus is only used to eliminate false positives from the data-flow analysis. Further information to generate test cases is not derived and a visualization is not conducted (Arzt et al., 2015).

Su et al. (Su et al., 2015) combine symbolic execution with static analysis and model-checking for C programs. A dynamic symbolic execution is utilized to automatically generate test data based on the data flow derived from a static analysis. For each identified DUC, the symbolic execution is started to find test data covering this DUC based on a search strategy that looks into the more promising control flow first. To reduce the amount of time trying to cover an unreachable DUC, model-checking is used additionally to identify unreachable DUCs (Su et al., 2015). However, by searching for every chain, paths are executed multiple times even if the chains are covered by the same input data. This is prevented by our approach of deriving the data flow during the symbolic execution. Moreover, Su et al. (Su et al., 2015) focus solely on the test data creation and do not mention any test data reduction approach or visualization.

In (Winkelmann et al., 2022), a dynamic data-flow analysis is used as a means for test case reduction for a test case generator based on symbolic execution. The data flow is tracked during the execution of paths. However, the focus lies on the symbolic execution and test case generation so that the resulting data flow is solely used for the reduction and not communicated or visualized to the user. Moreover, the data flow is only derived for the symbolic execution and not for existing test cases. In consequence, it is not indicated which DUCs were already covered by the existing test suite. Finally, the approach is hardwired into a custom JVM that was adapted to allow for symbolic execution. In contrast, both Mulib and Dacite, are standalone tools. It was demonstrated that two standard tools following the program instrumentation and transformation paradigm can be integrated with a relatively small amount of effort, without affecting their generality.

7 CONCLUSION AND FUTURE WORK

The dynamic data-flow analysis of Dacite is only able to identify DUCs that were covered by the given tests so far. In this paper, we have combined Dacite with the symbolic execution of Mulib to derive DUCs that have not been covered yet. To achieve this, a symbolic driver is generated automatically based on given tests and given to the user. With this driver, the symbolic execution is triggered within Mulib, during which DUCs are collected by Dacite. Both Dacite and Mulib are adapted to facilitate the interactions without affecting their generality. Moreover, the delta of derived DUCs and already covered DUCs, the unaccounted-for DUCs are visualized to the user integrated into common IDEs. Additionally, test cases can be generated for those unaccounted-for DUCs.

In the future, we plan to further optimize the symbolic execution for deriving the unaccounted-for DUCs. One way would be to prune the search region based on the existing JUnit test cases and consequently the already covered DUCs for those paths. If, for instance, a region of the program cannot contain any more DUCs, these paths do not need to be regarded any longer during symbolic execution. This

serves as a countermeasure against the well-known problem of path explosion (Cadar and Sen, 2013).

REFERENCES

Allen, F. E. and Cocke, J. (1976). A program data flow analysis procedure. *Communications of the ACM*, 19(3):137.

Arzt, S., Rasthofer, S., Hahn, R., and Bodden, E. (2015). Using targeted symbolic execution for reducing false-positives in dataflow analysis. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State of the Art in Program Analysis*, pages 1–6.

Beyer, D. (2021). Software verification: 10th comparative evaluation (sv-comp 2021). *Proc. TACAS (2). LNCS*, 12652.

Bluemke, I. and Rembiszewski, A. (2009). Dataflow testing of java programs with dfc. In *IFIP Central and East European Conference on Software Engineering Techniques*, pages 215–228. Springer.

Cadar, C. and Sen, K. (2013). Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90.

de Araujo, R. P. A. and Chaim, M. L. (2014). Data-flow testing in the large. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 81–90. IEEE.

Denaro, G., Pezze, M., and Vivanti, M. (2014). On the right objectives of data flow testing. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 71–80. IEEE.

Frankl, P. G. and Weiss, S. N. (1993). An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787.

Frankl, P. G. and Weyuker, E. J. (1988). An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498.

Hemmati, H. (2015). How effective are code coverage criteria? In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 151–156. IEEE.

Khurshid, S., Păsăreanu, C. S., and Visser, W. (2003). Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'03, page 553–568, Berlin, Heidelberg. Springer-Verlag.

Majchrzak, T. A. and Kuchen, H. (2009). Automated test case generation based on coverage analysis. In *2009 Third IEEE International Symposium on Theoretical Aspects of Software Engineering*, pages 259–266.

Microsoft Corporation (2023a). Language server extension guide. https://code.visualstudio.com/api/language-extensions/language-server-extension-guide. Last accessed September 01, 2023.

Microsoft Corporation (2023b). Language Server Protocol Specification - 3.17. https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification. Last accessed September 01, 2023.

Pande, H. D., Landi, W. A., and Ryder, B. G. (1994). Interprocedural def-use associations for c systems with single level pointers. *IEEE Transactions on Software Engineering*, 20(5):385–403.

Ribeiro, H. L., de Araujo, P. R., Chaim, M. L., de Souza, H. A., and Kon, F. (2019). Evaluating data-flow coverage in spectrum-based fault localization. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11. IEEE.

Su, T., Fu, Z., Pu, G., He, J., and Su, Z. (2015). Combining symbolic execution and model checking for data flow testing. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 654–665. IEEE.

Troost, L. and Kuchen, H. (2022). A comprehensive dynamic data flow analysis of object-oriented programs. In *Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE*, pages 267–274. INSTICC, SciTePress.

Troost, L., Neugebauer, J., and Kuchen, H. (2023). Visualizing dynamic data-flow analysis of object-oriented programs based on the language server protocol. In *Proceedings of the 18th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE,*, pages 77–88. INSTICC, SciTePress.

Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. (2010). Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, CASCON '10, page 214–224, USA. IBM Corp.

Visser, W., Păsăreanu, C. S., and Khurshid, S. (2004). Test input generation with java pathfinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107.

Winkelmann, H., Dageförde, J. C., and Kuchen, H. (2021). Constraint-logic object-oriented programming with free arrays. In Hanus, M. and Sacerdoti Coen, C., editors, *Functional and Constraint Logic Programming*, pages 129–144, Cham. Springer International Publishing.

Winkelmann, H. and Kuchen, H. (2022). Constraint-logic object-oriented programming on the java virtual machine. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, SAC '22, page 1258–1267, New York, NY, USA. Association for Computing Machinery.

Winkelmann, H. and Kuchen, H. (2023). Constraint-logic object-oriented programming with free arrays of reference-typed elements via symbolic aliasing. In *Proceedings of the 18th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE*, pages 412–419. INSTICC, SciTePress.

Winkelmann, H., Troost, L., and Kuchen, H. (2022). Constraint-logic object-oriented programming for test case generation. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, SAC '22, page 1499–1508. ACM.